2 Your first application

This chapter covers

- Creating your first ASP.NET Core web application
- Running your application
- Understanding the components of your application

After reading chapter 1, you should have a general idea of how ASP.NET Core applications work and when you should use them. You should have also set up a development environment you can use to start building applications.

> **TIP** See appendix A for guidance on installing the .NET 5.0 SDK and choosing an editor/IDE.

In this chapter, you'll dive right in by creating your first web app. You'll get to kick the tires and poke around a little to get a feel for how it works, and in later chapters I'll show you how to go about customizing and building your own applications.

As you work through this chapter, you should begin to get a grasp of the various components that make up an ASP.NET Core application, as well as an understanding of the general application-building process. Most applications you create will start from a similar *template*, so it's a good idea to get familiar with the setup as soon as possible.

DEFINITION A *template* provides the basic code required to build an application. You can use a template as the starting point for building your own apps.

I'll start by showing you how to create a basic ASP.NET Core application using one of the Visual Studio templates. If you're using other tooling, such as the .NET CLI, you'll have similar templates available. I use Visual Studio 2019 and ASP.NET Core 5.0 with .NET 5.0 in this chapter, but I also provide tips for working with the .NET CLI.

TIP You can view the application code for this chapter in the GitHub repository for the book at https://github.com/andrewlock/asp-dot-net-core-in-action-2e.

Once you've created your application, I'll show you how to restore all the necessary dependencies, compile your application, and run it to see the HTML output. The application will be simple in some respects—it will only have two different pages—but it'll be a fully configured ASP.NET Core application.

Having run your application, the next step will be to understand what's going on! We'll take a journey through all the major parts of an ASP.NET Core application, looking at how to configure the web server, the middleware pipeline, and HTML generation, among other things. We won't go into detail at this stage, but you'll get a feel for how they all work together to create a complete application.

We'll begin by looking at the plethora of files created when you start a new project, and you'll learn how a typical ASP.NET Core application is laid out. In particular, I'll focus on the Program.cs and Startup.cs files. Virtually the entire configuration of your application takes place in these two files, so it's good to get to grips with what they're for and how they're used. You'll see how to define the middleware pipeline for your application and how you can customize it.

Finally, you'll see how the app generates HTML in response to a request, looking at each of the components that make up the Razor Pages endpoint. You'll see how it controls what code is run in response to a request and how to define the HTML that should be returned for a particular request.

At this stage, don't worry if you find parts of the project confusing or complicated; you'll be exploring each section in detail as you move through the book. By the end of the chapter, you should have a basic understanding of how ASP.NET Core applications are put together, from when your application is first run to when a response is generated. Before we begin, though, we'll review how ASP.NET Core applications handle requests.

2.1 A brief overview of an ASP.NET Core application

In chapter 1, I described how a browser makes an HTTP request to a server and receives a response, which it uses to render HTML on the page. ASP.NET Core allows you to dynamically generate that HTML depending on the particulars of the request, so that you can, for example, display different data depending on the current logged-in user.

Say you want to create a web app to display information about your company. You could create a simple ASP.NET Core app to achieve this; then, later, you could add dynamic features to your app. Figure 2.1 shows how the application would handle a request for a page in your application.

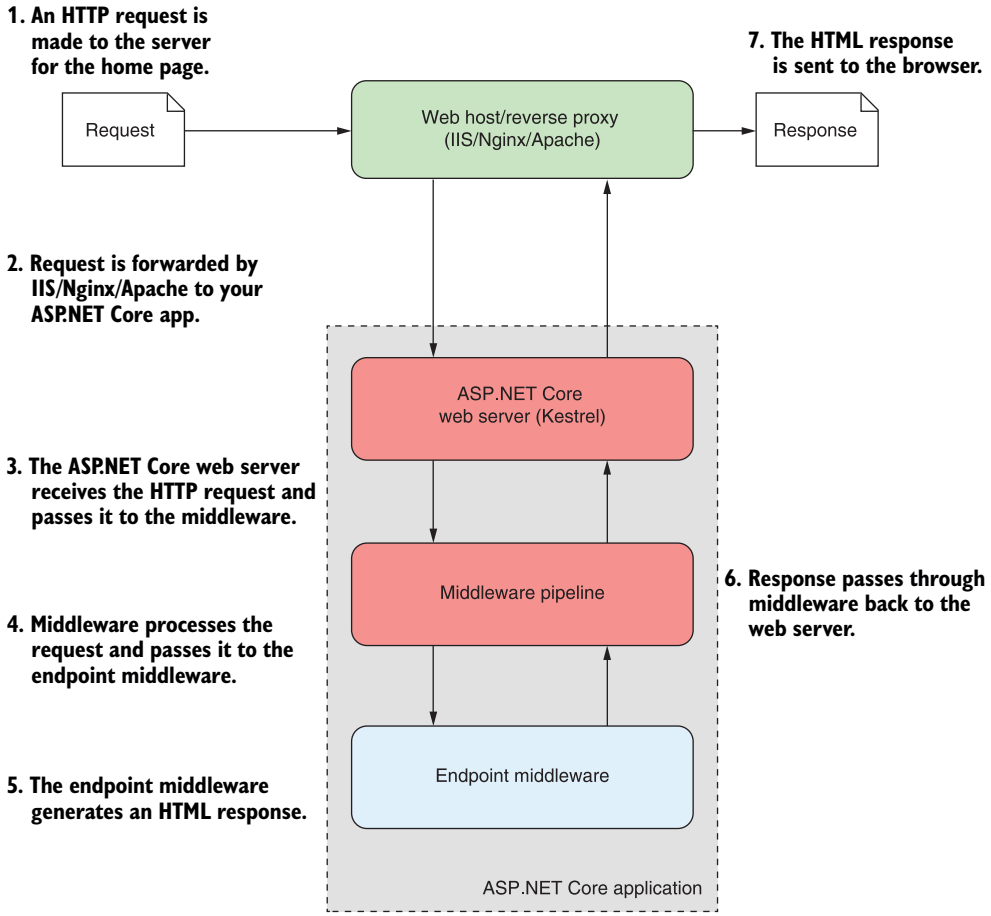

Figure 2.1 An overview of an ASP.NET Core application. The ASP.NET Core application receives an incoming HTTP request from the browser. Every request passes to the middleware pipeline, which potentially modifies it and then passes it to the endpoint middleware at the end of the pipeline to generate a response. The response passes back through the middleware to the server, and finally out to the browser.

Much of this diagram should be familiar to you from figure 1.8 in chapter 1; the request and response, the reverse proxy, and the ASP.NET Core web server are all still there, but you'll notice that I've expanded the ASP.NET Core application itself to

show the middleware pipeline and the endpoint middleware. This is the main custom part of your app that goes into generating the response from a request.

The first port of call after the reverse proxy forwards a request is the ASP.NET Core web server, which is the default cross-platform Kestrel server. Kestrel takes the raw incoming network request and uses it to generate an `HttpContext` object that the rest of the application can use.

The HttpContext object

The `HttpContext` constructed by the ASP.NET Core web server is used by the application as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with it and stored in it. This could include properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details and passes it on to the rest of the application.

NOTE Kestrel isn't the only HTTP server available in ASP.NET Core, but it's the most performant and is cross-platform. I'll only refer to Kestrel throughout the book. The main alternative, HTTP.sys, only runs on Windows and can't be used with IIS.[1]

Kestrel is responsible for receiving the request data and constructing a C# representation of the request, but it doesn't attempt to generate a response directly. For that, Kestrel hands the `HttpContext` to the middleware pipeline found in every ASP.NET Core application. This is a series of components that process the incoming request to perform common operations such as logging, handling exceptions, or serving static files.

NOTE You'll learn about the middleware pipeline in detail in the next chapter.

At the end of the middleware pipeline is the *endpoint* middleware. This middleware is responsible for calling the code that generates the final response. In most applications that will be an MVC or Razor Pages block.

Razor Pages are responsible for generating the HTML that makes up the pages of a typical ASP.NET Core web app. They're also typically where you find most of the business logic of your app, calling out to various services in response to the data contained in the original request. Not every app needs an MVC or Razor Pages block, but it's typically how you'll build most apps that display HTML to a user.

NOTE I'll cover Razor Pages and MVC controllers in chapter 4, including how to choose between them. I cover generating HTML in chapters 7 and 8.

[1] If you want to learn more about HTTP.sys, the documentation describes the server and how to use it: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/httpsys.

Most ASP.NET Core applications follow this basic architecture, and the example in this chapter is no different. First you'll see how to create and run your application, and then we'll look at how the code corresponds to the outline in figure 2.1. Without further ado, let's create an application!

2.2 Creating your first ASP.NET Core application

You can start building applications with ASP.NET Core in many different ways, depending on the tools and operating system you're using. Each set of tools will have slightly different templates, but they have many similarities. The example used throughout this chapter is based on a Visual Studio 2019 template, but you can easily follow along with templates from the .NET CLI or Visual Studio for Mac.

REMINDER This chapter uses Visual Studio 2019 and ASP.NET Core 5.0 with .NET 5.0.

Getting an application up and running typically involves four basic steps, which we'll work through in this chapter:

1. *Generate*—Create the base application from a template to get started.
2. *Restore*—Restore all the packages and dependencies to the local project folder using NuGet.
3. *Build*—Compile the application and generate all the necessary assets.
4. *Run*—Run the compiled application.

Visual Studio and the .NET CLI include many ASP.NET Core templates for building different types of applications. For example,

- *Razor Pages web application*—Razor Pages applications generate HTML on the server and are designed to be viewed by users in a web browser directly.
- *MVC (Model-View-Controller) application*—MVC applications are similar to Razor Pages apps in that they generate HTML on the server and are designed to be viewed by users directly in a web browser. They use traditional MVC controllers instead of Razor Pages.
- *Web API application*—Web API applications return data in a format that can be consumed by single-page applications (SPAs) and APIs. They are typically used in conjunction with client-side applications like Angular and React.js or mobile applications.

We will look at each of these application types in this book, but in this chapter we will focus on the Razor Pages template.

2.2.1 Using a template to get started

Using a template can quickly get you up and running with an application, automatically configuring many of the fundamental pieces. Both Visual Studio and the .NET CLI come with a number of standard templates for building web applications, console applications, and class libraries.

TIP In .NET, a *project* is a unit of deployment, which will be compiled into a .dll file or an executable, for example. Each separate app is a separate project. Multiple projects can be built and developed at once in a *solution*.

To create your first web application, open Visual Studio and perform the following steps:

1. Choose Create a New Project from the splash screen, or choose File > New > Project from the main Visual Studio screen.
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template, as shown in figure 2.2. Click Next.

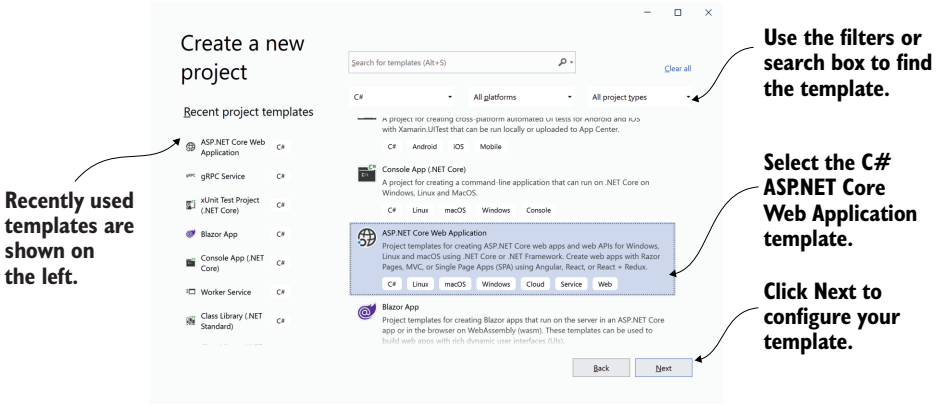

Figure 2.2 The new project dialog box. Select the C# ASP.NET Core Web Application template from the list on the right side. When you next create a new project, you can select from the recent templates list on the left.

3. On the next screen, enter a project name, location, and solution name, and click Create, as shown in figure 2.3. For example, use WebApplication1 as both the project and solution name.
4. On the following screen (figure 2.4):
   - Ensure .NET Core is selected.
   - Select ASP.NET Core 5.0. If this option isn't available, ensure you have .NET 5.0 installed. See appendix A for details on configuring your environment.
   - Select ASP.NET Core Web App to create a Razor Pages web application.
   - Ensure No Authentication is specified. You'll learn how to add users to your app in chapter 14.
   - Ensure Configure for HTTPS is checked.
   - Ensure Enable Docker Support is unchecked.
   - Click Create.

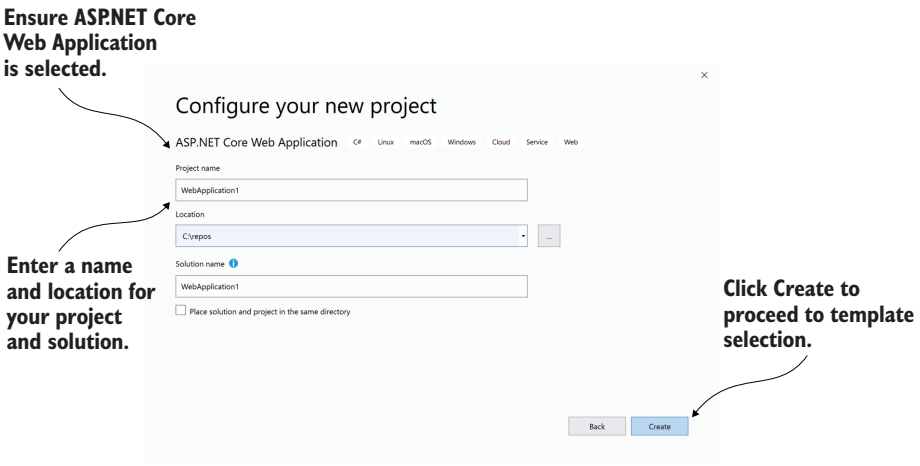

Figure 2.3 The Configure Your New Project dialog box. To create a new .NET 5.0 application, select ASP.NET Core Web Application from the template screen. On the following screen, enter a project name, location, and solution name, and click Create.

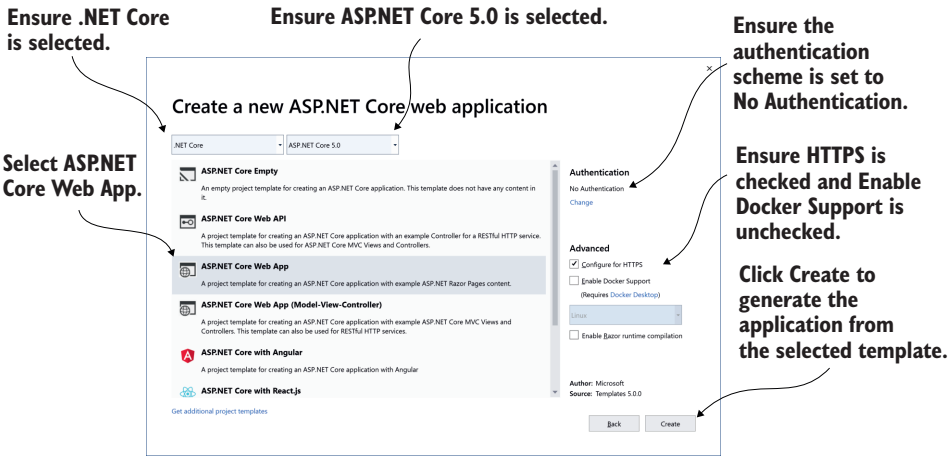

Figure 2.4 The web application template screen. This screen follows on from the Configure Your Project dialog box and lets you customize the template that will generate your application. For this starter project, you'll create a Razor Pages web application without authentication.

5 Wait for Visual Studio to generate the application from the template. Once Visual Studio has finished, you'll be presented with an introductory page about ASP.NET Core, and you should be able to see that Visual Studio has created and added a number of files to your project, as shown in figure 2.5.

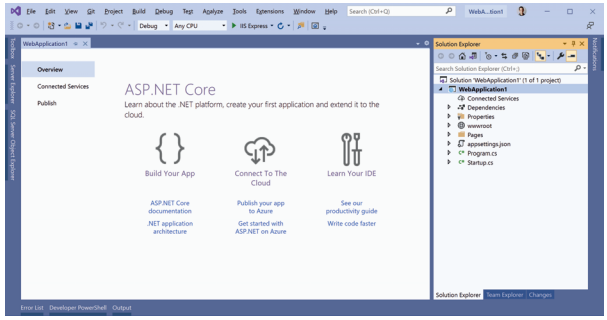

Figure 2.5 Visual Studio after creating a new ASP.NET Core application from a template. The Solution Explorer shows your newly created project. The introductory page has helpful links for learning about ASP.NET Core.

If you're not using Visual Studio, you can create a similar template using the .NET CLI. Create a folder to hold your new project. Open a PowerShell or cmd prompt in the folder (on Windows) or a terminal session (on Linux or macOS) and run the commands in the following listing.

Listing 2.1 Creating a new Razor Page application with the .NET CLI

```
dotnet new sln -n WebApplication1
dotnet new webapp -o WebApplication1
dotnet sln add WebApplication1
```

Create a solution file called WebApplication1 in the current folder.

Add the new project to the solution file.

Create a Razor Pages project in a subfolder, WebApplication1.

Whether you use Visual Studio or the .NET CLI, you now have the basic files required to build and run your first ASP.NET Core application.

2.2.2 Building the application

At this point, you have most of the files necessary to run your application, but you've got two steps left. First, you need to ensure all the dependencies used by your project are copied to your local directory, and second, you need to compile your application so that it can be run.

The first of these steps isn't strictly necessary, as both Visual Studio and the .NET CLI automatically restore packages when they first create your project, but it's good to know what's going on. In earlier versions of the .NET CLI, before 2.0, you needed to manually restore packages using `dotnet restore`.

You can compile your application by choosing Build > Build Solution, by using the shortcut Ctrl-Shift-B, or by running `dotnet build` from the command line. If you build from Visual Studio, the output window shows the progress of the build, and assuming everything is hunky dory, will compile your application, ready for running.

You can also run the `dotnet build` console commands from the Package Manager Console in Visual Studio.

TIP Visual Studio and the .NET CLI tools will automatically build your application when you run it if they detect that a file has changed, so you generally won't need to explicitly perform this step yourself.

NuGet packages and the .NET command-line interface

One of the foundational components of .NET 5.0 cross-platform development is the .NET command-line interface (CLI). This provides several basic commands for creating, building, and running .NET 5.0 applications. Visual Studio effectively calls these automatically, but you can also invoke them directly from the command line if you're using a different editor. The most common commands used during development are

- `dotnet restore`
- `dotnet build`
- `dotnet run`

Each of these commands should be run inside your project folder and will act on that project alone.

Most ASP.NET Core applications have dependencies on various external libraries, which are managed through the NuGet package manager. These dependencies are listed in the project, but the files of the libraries themselves aren't included. Before you can build and run your application, you need to ensure there are local copies of each dependency in your project folder. The first command, `dotnet restore`, ensures your application's NuGet dependencies are copied to your project folder.

ASP.NET Core projects list their dependencies in the project's .csproj file. This is an XML file that lists each dependency as a `PackageReference` node. When you run `dotnet restore`, it uses this file to establish which NuGet packages to download and copy to your project folder. Any dependencies listed are available for use in your application.

The restore process typically happens implicitly when you build or run your application, but it can sometimes be useful to run it explicitly, in continuous-integration build pipelines, for example.

You can compile your application using `dotnet build`. This will check for any errors in your application and, if there are no issues, will produce output that can be run using `dotnet run`.

Each command contains a number of switches that can modify its behavior. To see the full list of available commands, run

```
dotnet --help
```

or to see the options available for a particular command, `new` for example, run

```
dotnet new --help
```

2.3 Running the web application

You're ready to run your first application, and there are a number of different ways to go about it. In Visual Studio, you can either click the green arrow on the toolbar next to IIS Express, or press the F5 shortcut. Visual Studio will automatically open a web browser window for you with the appropriate URL and, after a second or two, you should be presented with your brand-new application, as shown in figure 2.6. Alternatively, you can run the application from the command line with the .NET CLI tools using `dotnet run`, and open the URL in a web browser manually, using the address provided on the command line.

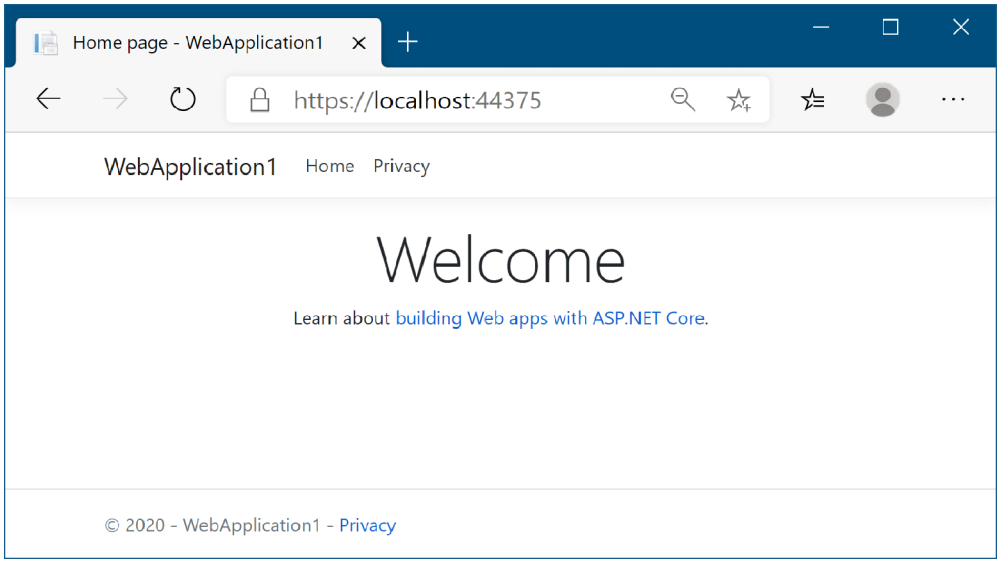

Figure 2.6 The home page of your new ASP.NET Core application. When you run it from Visual Studio, IIS Express chooses a random port by default. If you're running from the command line with `dotnet run`, your application will be available at http://localhost:5000 and https://localhost:5001.

> **TIP** The first time you run the application from Visual Studio, you will be prompted to install the development certificate. Doing so ensures your browser doesn't display warnings about an invalid certificate.[2] See chapter 18 for more about HTTPS certificates.

By default, this page shows a simple Welcome banner and a link to the official Microsoft documentation for ASP.NET Core. At the top of the page are two links: Home

[2] You can install the development certificate on Windows and macOS. For instructions on trusting the certificate on Linux, see your distribution's instructions. Not all browsers (Mozilla Firefox, for example) use the certificate store, so follow your browser's guidelines for trusting the certificate. If you still have difficulties, see the troubleshooting tips at http://mng.bz/1rmy.

and Privacy. The Home link is the page you're currently on. Clicking Privacy will take you to a new page, as shown in figure 2.7. As you'll see shortly, you can use Razor Pages in your application to define these two pages and build the HTML they display.

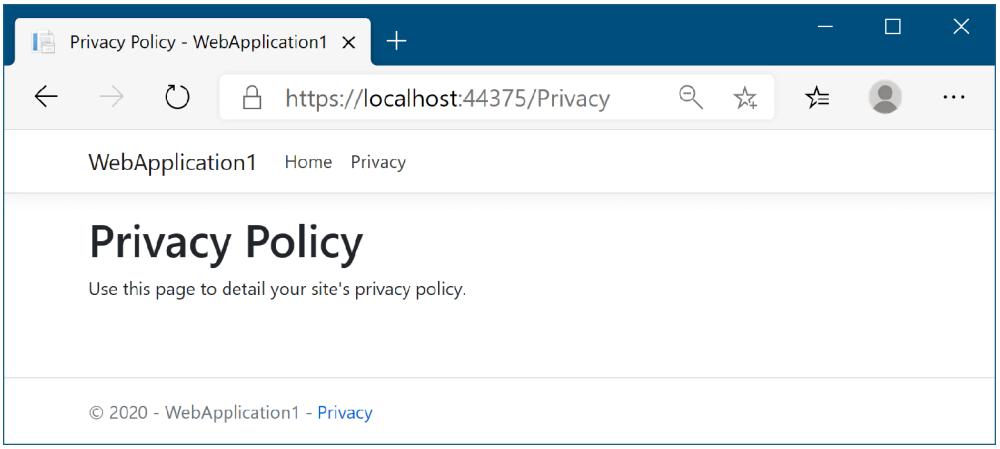

Figure 2.7 The Privacy page of your application. You can navigate between the two pages of the application using the Home and Privacy links in the application's header. The app generates the content of the pages using Razor Pages.

At this point, you need to notice a couple of things. First, the header containing the links and the application title, "WebApplication1," is the same on both pages. Second, the title of the page, as shown in the tab of the browser, changes to match the current page. You'll see how to achieve these features in chapter 7, when we discuss the rendering of HTML using Razor templates.

> **NOTE** You can only view the application on the same computer that is running it at the moment; your application isn't exposed to the internet yet. You'll learn how to publish and deploy your application in chapter 16.

There isn't any more to the user experience of the application at this stage. Click around a little and, once you're happy with the behavior of the application, roll up your sleeves—it's time to look at some code!

2.4 Understanding the project layout

When you're new to a framework, creating an application from a template like this can be a mixed blessing. On the one hand, you can get an application up and running quickly, with little input required on your part. Conversely, the number of files can sometimes be overwhelming, leaving you scratching your head working out where to start. The basic web application template doesn't contain a huge number of files and folders, as shown in figure 2.8, but I'll run through the major ones to get you oriented.

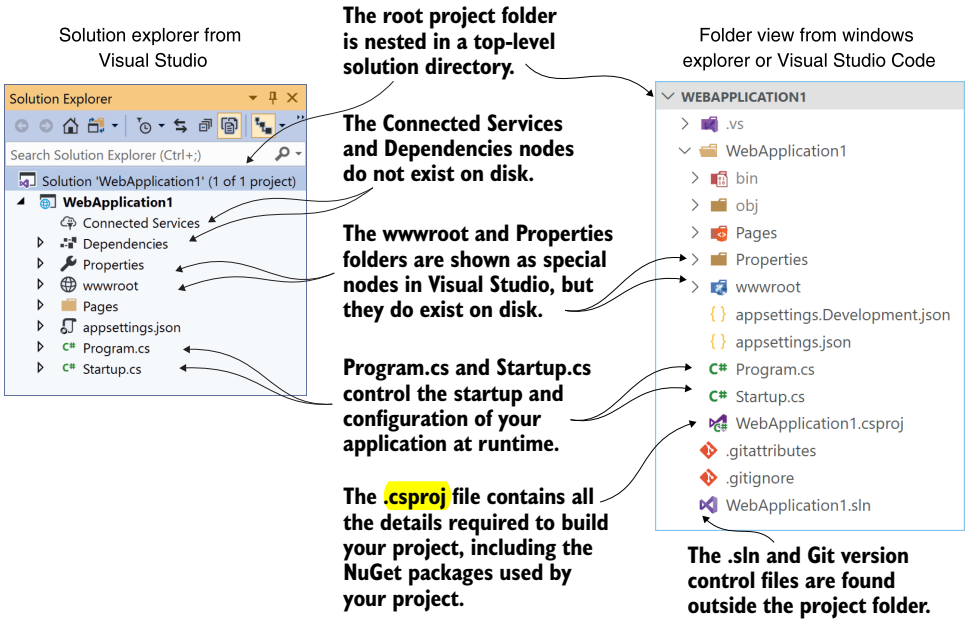

Figure 2.8 The Solution Explorer and folder on disk for a new ASP.NET Core application. The Solution Explorer also displays the Connected Services and Dependencies nodes, which list NuGet and other dependencies, though the folders themselves don't exist on disk.

The first thing to notice is that the main project, WebApplication1, is nested in a top-level directory with the name of the solution, which is also WebApplication1 in this case. Within this top-level folder, you'll also find the solution (.sln) file for use by Visual Studio and files related to Git version control,[3] though these are hidden in Visual Studio's Solution Explorer view.

> **NOTE** Visual Studio uses the concept of a solution to work with multiple projects. The example solution only consists of a single project, which is listed in the .sln file. If you use a CLI template to create your project, you won't have a .sln or Git files unless you generate them explicitly using additional .NET CLI templates.

Inside the solution folder, you'll find your project folder, which in turn contains three subfolders—Pages, Properties, and wwwroot. Pages (unsurprisingly) contains the Razor Pages files you'll use to build your application. The Properties folder contains a single file, launchSettings.json, which controls how Visual Studio will run and

[3] The Git files will only be added if you choose Add to Source Control in Visual Studio. You don't have to use Git, but I strongly recommend using some sort of version control when you build applications. If you're somewhat familiar with Git, but still find it a bit daunting, and a rebase terrifying, I highly recommend reading "Think Like (a) Git," http://think-like-a-git.net/. It helped me achieve Git enlightenment.

debug the application. The wwwroot folder is special, in that it's the only folder in your application that browsers are allowed to directly access when browsing your web app. You can store your CSS, JavaScript, images, or static HTML files in here and browsers will be able to access them. They won't be able to access any file that lives outside of wwwroot.

Although the wwwroot and Properties folders exist on disk, you can see that Solution Explorer shows them as special nodes, out of alphabetical order, near the top of your project. You've got two more special nodes in the project, Dependencies and Connected Services, but they don't have corresponding folders on disk. Instead, they show a collection of all the dependencies, such as NuGet packages, and remote services that the project relies on.

In the root of your project folder, you'll find two JSON files: appsettings.json and appsettings.Development.json. These provide configuration settings that are used at runtime to control the behavior of your app.

The most important file in your project is WebApplication1.csproj, as it describes how to build your project. Visual Studio doesn't explicitly show the .csproj file, but you can edit it if you double-click the project name in Solution Explorer. We'll have a closer look at this project file in the next section.

Finally, Visual Studio shows two C# files in the project folder—Program.cs and Startup.cs. In sections 2.6 and 2.7, you'll see how these fundamental classes are responsible for configuring and running your application.

2.5 The .csproj project file: Defining your dependencies

The .csproj file is the project file for .NET applications and contains the details required for the .NET tooling to build your project. It defines the type of project being built (web app, console app, or library), which platform the project targets (.NET Core 3.1, .NET 5.0, and so on), and which NuGet packages the project depends on.

The project file has been a mainstay of .NET applications, but in ASP.NET Core it has had a facelift to make it easier to read and edit. These changes include

- *No GUIDs*—Previously, globally unique identifiers (GUIDs) were used for many things, but now they're rarely used in the project file.
- *Implicit file includes*—Previously, every file in the project had to be listed in the .csproj file for it to be included in the build. Now, files are automatically compiled.
- *No paths to NuGet package .dll files*—Previously, you had to include the path to the .dll files contained in NuGet packages in the .csproj, as well as listing the dependencies in a packages.config file. Now you can reference the NuGet package directly in your .csproj, and you don't need to specify the path on disk.

All of these changes combine to make the project file far more compact than you'll be used to from previous .NET projects. The following listing shows the entire .csproj file for your sample app.

Listing 2.2 The .csproj project file, showing SDK, target framework, and references

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

The SDK attribute specifies the type of project you're building.

The TargetFramework is the framework you'll run on, in this case, .NET 5.0.

For simple applications, you probably won't need to change the project file much. The `Sdk` attribute on the `Project` element includes default settings that describe how to build your project, whereas the `TargetFramework` element describes the framework your application will run on. For .NET Core 3.1 projects, this will have the `netcoreapp3.1` value; if you're running on .NET 5.0, this will be `net5.0`.

> **TIP** With the new csproj style, Visual Studio users can double-click a project in Solution Explorer to edit the .csproj file without having to close the project first.

The most common changes you'll make to the project file are to add additional NuGet packages using the `PackageReference` element. By default, your app doesn't reference any NuGet packages at all.

Using NuGet libraries in your project

Even though all apps are unique in some way, they also share common requirements. For example, most apps need to access a database or manipulate JSON or XML formatted data. Rather than having to reinvent that code in every project, you should use existing reusable libraries.

NuGet is the library package manager for .NET, where libraries are packaged into *NuGet packages* and published to https://nuget.org. You can use these packages in your project by referencing the unique package name in your .csproj file. These make the package's namespace and classes available in your code files. You can publish (and host) NuGet packages to repositories other than https://nuget.org—see https://docs.microsoft.com/nuget for details.

You can add a NuGet reference to your project by running `dotnet add package <packagename>` from inside the project folder. This updates your project file with a `<PackageReference>` node and restores the NuGet package for your project. For example, to install the popular Newtonsoft.Json library, you would run

```
dotnet add package Newtonsoft.Json
```

This adds a reference to the latest version of the library to your project file, as shown next, and makes the Newtonsoft.Json namespace available in your source code files.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
```

```
  <ItemGroup>
    <PackageReference Include="NewtonSoft.Json" Version="12.0.3" />
  </ItemGroup>
</Project>
```

If you're using Visual Studio, you can manage packages with the NuGet Package Manager by right-clicking the solution name or a project and choosing Manage NuGet Packages.

As a point of interest, there's no officially agreed upon pronunciation for NuGet. Feel free to use the popular "noo-get" or "nugget" styles, or if you're feeling especially posh, "noo-jay"!

The simplified project file format is much easier to edit by hand than previous versions, which is great if you're developing cross-platform. But if you're using Visual Studio, don't feel like you have to take this route. You can still use the GUI to add project references, exclude files, manage NuGet packages, and so on. Visual Studio will update the project file itself, as it always has.

TIP For further details on the changes to the csproj format, see the documentation at http://mng.bz/PPGg.

The project file defines everything Visual Studio and the .NET CLI need to build your app. Everything, that is, except the code! In the next section we'll take a look at the entry point for your ASP.NET Core application—the Program.cs class.

2.6 The Program class: Building a web host

All ASP.NET Core applications start in the same way as .NET Console applications—with a Program.cs file. This file contains a `static void Main` function, which is a standard characteristic of console apps. This method must exist and is called whenever you start your web application.

TIP .NET 5.0 and C# 9 introduced "top-level statements," which implicitly create the `Main` entry point. I don't use top-level statements in this book, but they are supported in ASP.NET Core 5.0. See the documentation for details: http://mng.bz/JDaP.

In ASP.NET Core applications, the `Main` entry point is used to build and run an `IHost` instance, as shown in the following listing, which shows the default Program.cs file. The `IHost` is the core of your ASP.NET Core application, containing the application configuration and the Kestrel server that listens for requests and sends responses.

Listing 2.3 The default Program.cs file configures and runs an `IWebHost`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args)
            .Build()
            .Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Create an IHostBuilder using the CreateHostBuilder method.

Build and return an instance of IHost from the IHostBuilder.

Run the IHost and start listening for requests and generating responses.

Create an IHostBuilder using the default configuration.

Configure the application to use Kestrel and listen to HTTP requests.

The Startup class defines most of your application's configuration.

The `Main` function contains all the basic initialization code required to create a web server and to start listening for requests. It uses an `IHostBuilder`, created by the call to `CreateDefaultBuilder`, to define how the `IHost` is configured, before instantiating the `IHost` with a call to `Build()`.

> **NOTE** You'll find this pattern of using a builder object to configure a complex object repeated throughout the ASP.NET Core framework. It's a useful technique for allowing users to configure an object, delaying its creation until all configuration has finished. It's one of the patterns described in the "Gang of Four" book, *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison Wesley, 1994).

Much of your app's configuration takes place in the `IHostBuilder` created by the call to `CreateDefaultBuilder`, but it delegates some responsibility to a separate class, `Startup`. The `Startup` class referenced in the generic `UseStartup<>` method is where you configure your app's services and define your middleware pipeline. In section 2.7, we'll spend a while delving into this crucial class.

At this point, you may be wondering why you need two classes for configuration: `Program` and `Startup`. Why not include all your app's configuration in one class or the other?

Figure 2.9 shows the typical split of configuration components between `Program` and `Startup`. Generally speaking, `Program` is where you configure the infrastructure of your application, such as the HTTP server, integration with IIS, and configuration sources. In contrast, `Startup` is where you define which components and features your application uses, and the middleware pipeline for your app.

The `Program` class for two different ASP.NET Core applications will generally be similar, but the `Startup` classes will often differ significantly (though they generally

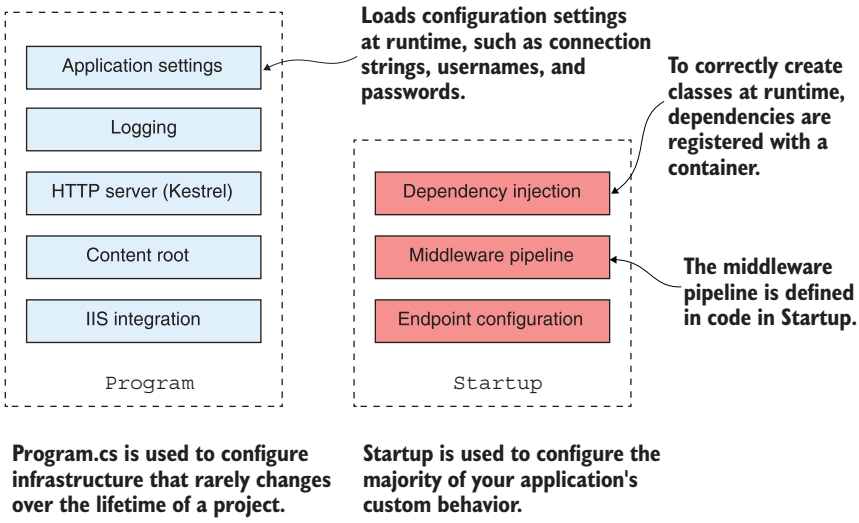

Figure 2.9 The difference in configuration scope for `Program` and `Startup`. `Program` is concerned with infrastructure configuration that will typically remain stable throughout the lifetime of the project. In contrast, you'll often modify `Startup` to add new features and to update application behavior.

follow a similar pattern, as you'll see shortly). You'll rarely find that you need to modify `Program` as your application grows, whereas you'll normally update `Startup` whenever you add additional features. For example, if you add a new NuGet dependency to your project, you'll normally need to update `Startup` to make use of it.

The `Program` class is where a lot of app configuration takes place, but in the default templates this is hidden inside the `CreateDefaultBuilder` method. `CreateDefaultBuilder` is a static helper method that simplifies the bootstrapping of your app by creating an `IHostBuilder` with some common configuration. In chapter 11 we'll peek inside this method and explore the configuration system, but for now it's enough to keep figure 2.9 in mind, and to be aware that you can completely change the `IHost` configuration if you need to.

The other helper method used by default is `ConfigureWebHostDefaults`. This uses a `WebHostBuilder` object to configure Kestrel to listen for HTTP requests.

Creating services with the generic host

It might seem strange that you must call `ConfigureWebHostDefaults` as well as `CreateDefaultBuilder`—couldn't we just have one method? Isn't handling HTTP requests the whole *point* of ASP.NET Core?

Well, yes and no! ASP.NET Core 3.0 introduced the concept of a *generic host*. This allows you to use much of the same framework as ASP.NET Core applications to write

(continued)

non-HTTP applications. These apps can be run as console apps or can be installed as Windows services (or as systemd daemons on Linux), to run background tasks or read from message queues, for example.

Kestrel and the web framework of ASP.NET Core builds *on top* of the generic host functionality introduced in ASP.NET Core 3.0. To configure a typical ASP.NET Core app, you configure the generic host features that are common across all apps; features such as configuration, logging, and dependency services. For web applications, you then also configure the services, such as Kestrel, that are necessary to handle web requests.

In chapter 22 you'll see how to build applications using the generic host to run scheduled tasks and build services.

Once the configuration of the `IHostBuilder` is complete, the call to `Build` produces the `IHost` instance, but the application still isn't handling HTTP requests yet. It's the call to `Run` that starts the HTTP server listening. At this point, your application is fully operational and can respond to its first request from a remote browser.

2.7 The Startup class: Configuring your application

As you've seen, `Program` is responsible for configuring a lot of the infrastructure for your app, but you configure some of your app's behavior in `Startup`. The `Startup` class is responsible for configuring two main aspects of your application:

- *Service registration*—Any classes that your application depends on for providing functionality—both those used by the framework and those specific to your application—must be registered so that they can be correctly instantiated at runtime.
- *Middleware and endpoints*—How your application handles and responds to requests.

You configure each of these aspects in its own method in `Startup`: service registration in `ConfigureServices`, and middleware configuration in `Configure`. A typical outline of `Startup` is shown in the following listing.

Listing 2.4 An outline of Startup.cs showing how each aspect is configured

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // method details
    }
    public void Configure(IApplicationBuilder app)
    {
        // method details
    }
}
```

Configure services by registering them with the IServiceCollection.

Configure the middleware pipeline for handling HTTP requests.

The IHostBuilder created in Program calls ConfigureServices and then Configure, as shown in figure 2.10. Each call configures a different part of your application, making it available for subsequent method calls. Any services registered in the ConfigureServices method are available to the Configure method. Once configuration is complete, an IHost is created by calling Build() on the IHostBuilder.

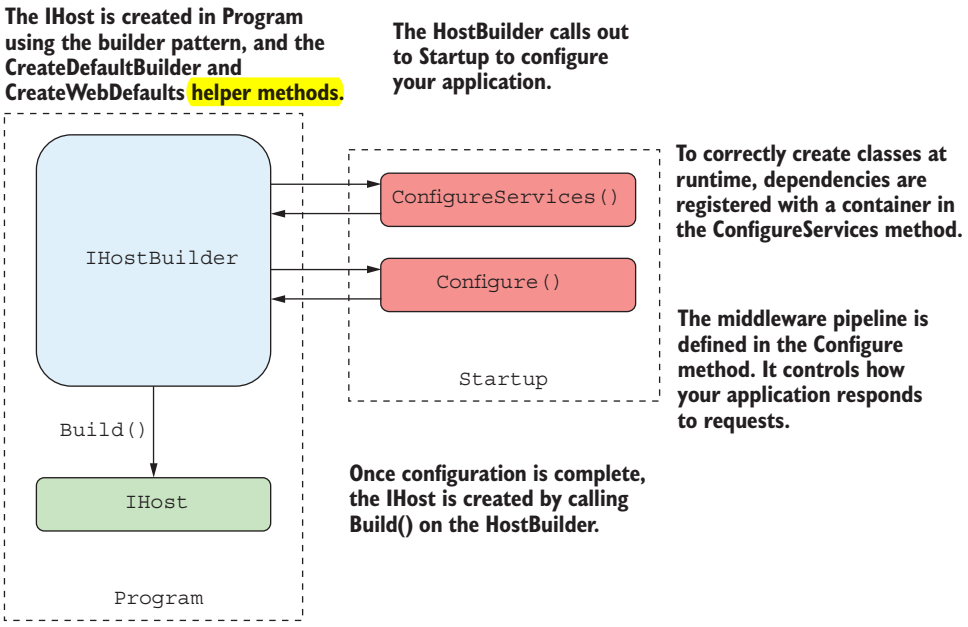

Figure 2.10 The IHostBuilder is created in Program.cs and calls methods on Startup to configure the application's services and middleware pipeline. Once configuration is complete, the IHost is created by calling Build() on the IHostBuilder.

An interesting point about the Startup class is that it doesn't implement an interface as such. Instead, the methods are invoked by using *reflection* to find methods with the predefined names of Configure and ConfigureServices. This makes the class more flexible and enables you to modify the signature of the method to accept additional parameters that are fulfilled automatically. I'll cover how this works in detail in chapter 10; for now it's enough to know that anything that's configured in ConfigureServices can be accessed by the Configure method.

> **DEFINITION** *Reflection* in .NET allows you to obtain information about types in your application at runtime. You can use reflection to create instances of classes at runtime and to invoke and access them.

Because the Startup class is fundamental to ASP.NET Core applications, the rest of section 2.7 walks you through both ConfigureServices and Configure to give you a

taste of how they're used. I won't explain them in detail (we have the rest of the book for that!), but you should keep in mind how they follow on from each other and how they contribute to the application's configuration as a whole.

2.7.1 Adding and configuring services

ASP.NET Core uses small, modular components for each distinct feature. This allows individual features to evolve separately, with only a loose coupling to others, and it's generally considered good design practice. The downside to this approach is that it places the burden on the consumer of a feature to correctly instantiate it. Within your application, these modular components are exposed as one or more *services* that are used by the application.

DEFINITION Within the context of ASP.Net Core, *service* refers to any class that provides functionality to an application. These could be classes exposed by a library or code you've written for your application.

For example, in an e-commerce app, you might have a `TaxCalculator` that calculates the tax due on a particular product, taking into account the user's location in the world. Or you might have a `ShippingCostService` that calculates the cost of shipping to a user's location. A third service, `OrderTotalCalculatorService`, might use both of these services to work out the total price the user must pay for an order. Each service provides a small piece of independent functionality, but you can combine them to create a complete application. This is known as the *single responsibility principle*.

DEFINITION The *single responsibility principle* (SRP) states that every class should be responsible for only a single piece of functionality—it should only need to change if that required functionality changes. It's one of the five main design principles promoted by Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices* (Pearson, 2013).

`OrderTotalCalculatorService` needs access to an instance of `ShippingCostService` and `TaxCalculator`. A naive approach to this problem is to use the `new` keyword and create an instance of a service whenever you need it. Unfortunately, this tightly couples your code to the specific implementation you're using and can completely undo all the good work achieved by modularizing the features in the first place. In some cases, it may break the SRP by making you perform initialization code in addition to using the service you created.

One solution to this problem is to make it somebody else's problem. When writing a service, you can declare your dependencies and let another class fill those dependencies for you. Your service can then focus on the functionality for which it was designed, instead of trying to work out how to build its dependencies.

This technique is called dependency injection or the Inversion of Control (IoC) principle, and it is a well-recognized *design pattern* that is used extensively.

DEFINITION *Design patterns* are solutions to common software design problems.

Typically, you'll register the dependencies of your application into a "container," which can then be used to create any service. This is true for both your own custom application services and the framework services used by ASP.NET Core. You must register each service with the container before it can be used in your application.

> **NOTE** I'll describe the dependency inversion principle and the IoC container used in ASP.NET Core in detail in chapter 10.

In an ASP.NET Core application, this registration is performed in the `ConfigureServices` method. Whenever you use a new ASP.NET Core feature in your application, you'll need to come back to this method and add in the necessary services. This is not as arduous as it sounds, as shown in the following listing, taken from the example application.

Listing 2.5 `Startup.ConfigureServices`: adding services to the IoC container

```
public class Startup
{
    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
}
```

You may be surprised that a complete Razor Pages application only includes a single call to add the necessary services, but the `AddRazorPages()` method is an extension method that encapsulates all the code required to set up the Razor Pages services. Behind the scenes, it adds various Razor services for rendering HTML, formatting services, routing services, and many more.

As well as registering framework-related services, this method is where you'd register any custom services you have in your application, such as the example `TaxCalculator` discussed previously. `IServiceCollection` is a list of every known service that your application will need to use. By adding a new service to it, you ensure that whenever a class declares a dependency on your service, the IoC container knows how to provide it.

With your services all configured, it's time to move on to the final configuration step: defining how your application responds to HTTP requests.

2.7.2 Defining how requests are handled with middleware

So far, in the `IHostBuilder` and `Startup` classes, you've defined the infrastructure of the application and registered your services with the IoC container. In the final configuration method of the `Startup` class, `Configure`, you define the middleware pipeline for the application, which defines how your app handles HTTP requests. Here's the `Configure` method for the template application.

Listing 2.6 `Startup.Configure`: defining the middleware pipeline

```
public class Startup
{
    public void Configure(
        IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();

        app.UseStaticFiles();

        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

IApplicationBuilder is used to build the middleware pipeline.

Other services can be accepted as parameters.

Different behavior when in development or production

Only runs in a development environment

Only runs in a production environment

Adds the static-file middleware

Adds the endpoint routing middleware, which determines which endpoint to execute

Adds the authorization middleware, which can block access to specific pages as required

Adds the endpoint middleware, which executes a Razor Page to generate an HTML response

As I described previously, middleware consists of small components that execute in sequence when the application receives an HTTP request. They can perform a whole host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors.

The `IApplicationBuilder` that's passed to the `Configure` method is used to define the order in which middleware executes. The order of the calls in this method is important, as the order in which they're added to the builder is the order in which they'll execute in the final pipeline. Middleware can only use objects created by previous middleware in the pipeline—it can't access objects created by later middleware.

> **WARNING** It's important that you consider the order of middleware when adding it to the pipeline. Middleware can only use objects created by earlier middleware in the pipeline.

You should also note that an `IWebHostEnvironment` parameter is used to provide different behavior when you're in a development environment. When you're running in development (when `EnvironmentName` is set to `"Development"`), the `Configure` method adds one piece of exception-handling middleware to the pipeline; in production, it adds a different one.

The `IWebHostEnvironment` object contains details about the current environment, as determined by `IHostBuilder` in `Program`. It exposes a number of properties:

- `ContentRootPath`—Location of the working directory for the app, typically the folder in which the application is running
- `WebRootPath`—Location of the `wwwroot` folder that contains static files
- `EnvironmentName`—Whether the current environment is a development or production environment

`IWebHostEnvironment` is already set by the time `Startup` is invoked; you can't change these values using the application settings in `Startup`. `EnvironmentName` is typically set externally by using an environment variable when your application starts.

> **NOTE** You'll learn about hosting environments and how to change the current environment in chapter 11.

In development, `DeveloperExceptionPageMiddleware` (added by the `UseDeveloperExceptionPage()` call) ensures that if your application throws an exception that isn't caught, you'll be presented with as much information as possible in the browser to diagnose the problem, as shown in figure 2.11. It's akin to the "yellow screen of death" in the previous version of ASP.NET, but this time it's white, not yellow.

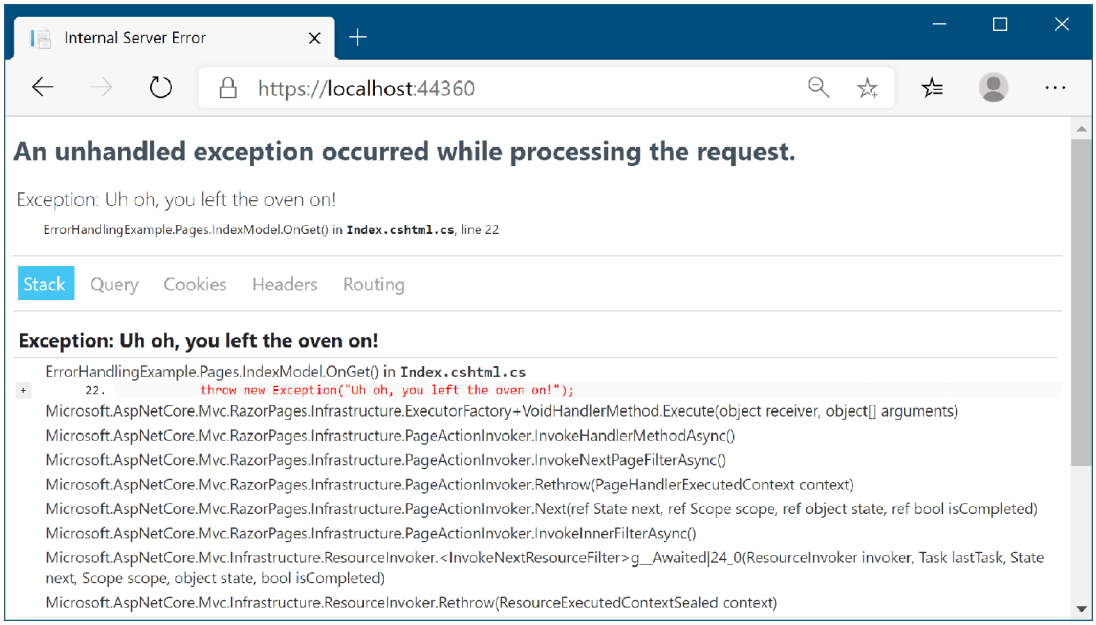

Figure 2.11 The developer exception page contains many different sources of information to help you diagnose a problem, including the exception stack trace and details about the request that generated the exception.

> **NOTE** The default templates also add `HstsMiddleware` in production, which sets security headers in your response, in line with industry best practices. See chapter 18 for details about this and other security-related middleware.

When you're running in a production environment, exposing this amount of data to users would be a big security risk. Instead, `ExceptionHandlerMiddleware` is registered so that if users encounter an exception in your method, they will be presented with a friendly error page that doesn't reveal the source of the problems. If you run the default template in production mode and trigger an error, you'll be presented with the message shown in figure 2.12 instead. Obviously, you'd need to update this page to be more visually appealing and more user-friendly, but at least it doesn't reveal the inner workings of your application.

The next piece of middleware added to the pipeline is `HttpsRedirectionMiddleware`, using this statement:

```
app.UseHttpsRedirection();
```

This ensures your application only responds to secure (HTTPS) requests and is an industry best practice. We'll look more at HTTPS in chapter 18.

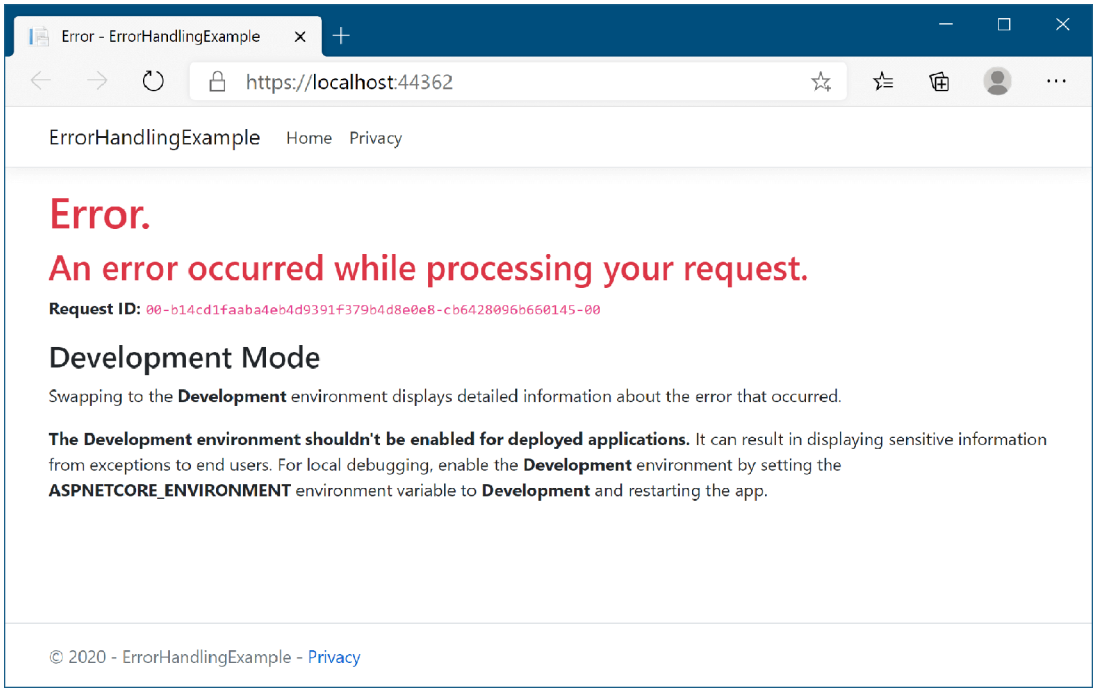

Figure 2.12 The default exception-handling page. In contrast to the developer exception page, this one doesn't reveal any details about your application to users. In reality, you'd update the message to something more user-friendly.

`StaticFileMiddleware` is added to the pipeline next with this statement:

```
app.UseStaticFiles();
```

This middleware is responsible for handling requests for static files such as CSS files, JavaScript files, and images. When a request arrives at the middleware, it checks to see if the request is for an existing file. If it is, the middleware returns the file. If not, the request is ignored and the next piece of middleware can attempt to handle the request. Figure 2.13 shows how the request is processed when a static file is requested. When

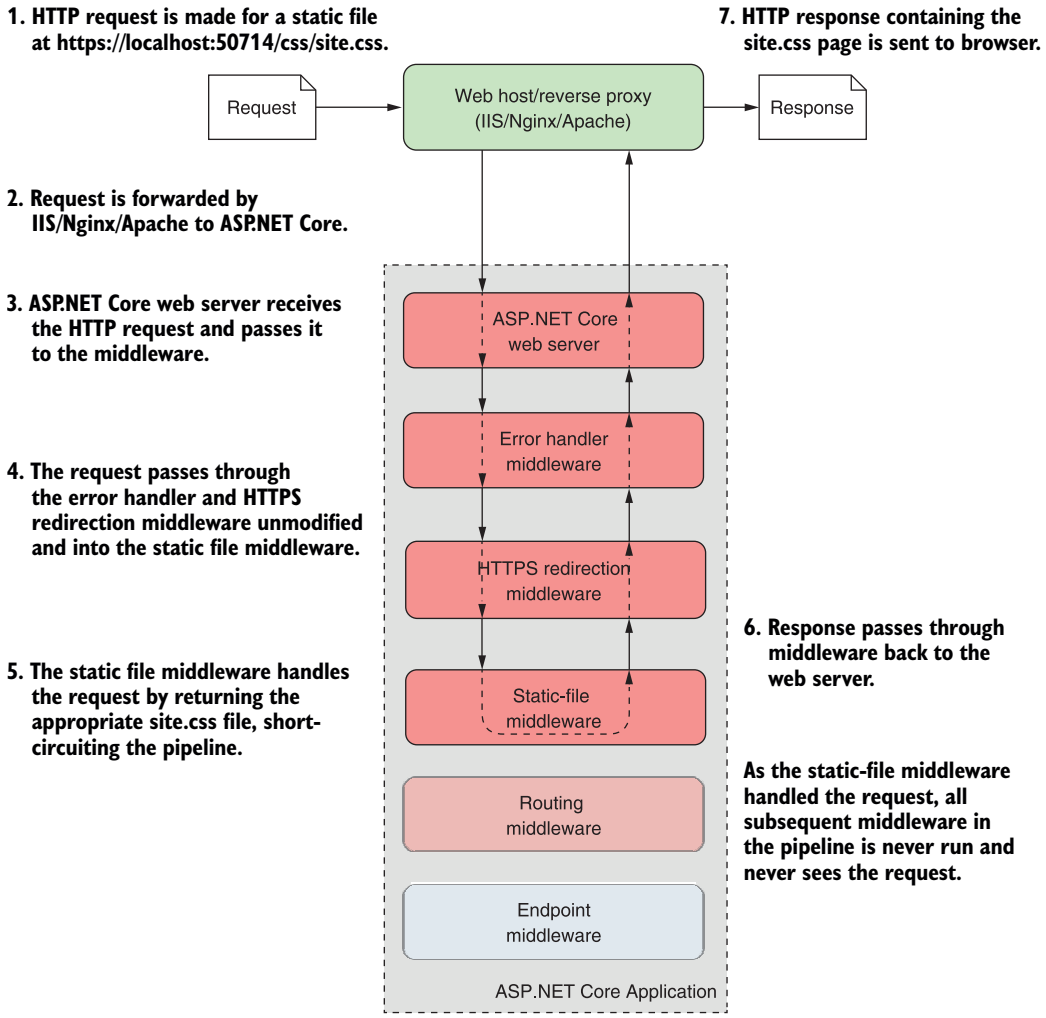

Figure 2.13 An overview of a request for a static file at `/css/site.css` for an ASP.NET Core application. The request passes through the middleware pipeline until it's handled by the static-file middleware. This returns the requested CSS file as the response, which passes back to the web server. The endpoint middleware is never invoked and never sees the request.

the static-file middleware handles a request, other middleware that comes later in the pipeline, such as the routing middleware or the endpoint middleware, won't be called at all.

Now we come to the most substantial pieces of middleware in the pipeline: the routing middleware and the endpoint middleware. Together, this pair of middleware are responsible for interpreting the request to determine which Razor Page to invoke, for reading parameters from the request, and for generating the final HTML. Very little configuration is required—you need only to add the middleware to the pipeline and specify that you wish to use Razor Page endpoints by calling `MapRazorPages`. For each request, the routing middleware uses the request's URL to determine which Razor Page to invoke. The endpoint middleware actually executes the Razor Page to generate the HTML response.

> **NOTE** The default templates also add the `AuthorizationMiddleware` *between* the routing middleware and the endpoint middleware. This allows the authorization middleware to decide whether to allow access *before* the Razor Page is executed. You'll learn more about this approach in chapter 5 on routing and chapter 15 on authorization.

Phew! You've finally finished configuring your application with all the settings, services, and middleware it needs. Configuring your application touches on a wide range of different topics that we'll delve into further throughout the book, so don't worry if you don't fully understand all the steps yet.

Once the application is configured, it can start handling requests. But *how* does it handle them? I've already touched on `StaticFileMiddleware`, which will serve the image and CSS files to the user, but what about the requests that require an HTML response? In the rest of this chapter, I'll give you a glimpse into Razor Pages and how they generate HTML.

2.8 Generating responses with Razor Pages

When an ASP.NET Core application receives a request, it progresses through the middleware pipeline until a middleware component can handle it, as you saw in figure 2.13. Normally, the final piece of middleware in a pipeline is the endpoint middleware. This middleware works with the routing middleware to match a request URL's path to a configured route, which defines which Razor Page to invoke.

> **DEFINITION** A path is the remainder of the request URL, once the domain has been removed. For example, for a request to www.microsoft.com/account/manage, the path is `/account/manage`.

Once a Razor Page has been selected, the routing middleware notes the selected Razor Page in the request's `HttpContext` and continues executing the middleware pipeline. Eventually the request will reach the endpoint middleware. The endpoint middleware executes the Razor Page to generate the HTML response and sends it back to the browser, as shown in figure 2.14.

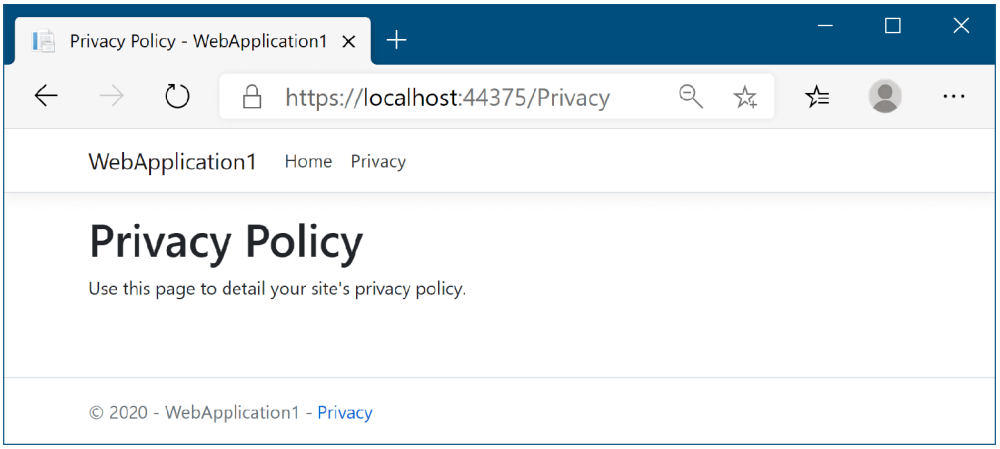

Figure 2.14 Rendering a Razor template to HTML. The Razor Page is selected based on the URL page `/Privacy` and is executed to generate the HTML.

In the next section we'll look at how Razor Pages generate HTML using the Razor syntax. After that we'll look at how you can use page handlers to add business logic and behavior to your Razor Pages.

2.8.1 Generating HTML with Razor Pages

Razor Pages are stored in .cshtml files (a portmanteau of .cs and .html) within the Pages folder of your project. In general, the routing middleware maps request URL paths to a single Razor Page by looking in the Pages folder of your project for a Razor Page with the same path. For example, you can see in figure 2.14 that the Privacy page of your app corresponds to the path `/Privacy` in the browser's address bar. If you look inside the Pages folder of your project, you'll find the Privacy.cshtml file, shown in the following listing.

Listing 2.7 The Privacy.cshtml Razor Page

```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p>Use this page to detail your site's privacy policy.</p>
```

Indicates that this is a Razor Page

Links the Razor Page to a specific PageModel

C# code that doesn't write to the response

HTML with dynamic C# values written to the response

Standalone, static HTML

Razor Pages use a templating syntax, called *Razor*, that combines static HTML with dynamic C# code and HTML generation. The `@page` directive on the first line of the

Razor Page is the most important. This directive must always be placed on the first line of the file, as it tells ASP.NET Core that the .cshtml file is a Razor Page. Without it, you won't be able to view your page correctly.

The next line of the Razor Page defines which `PageModel` in your project the Razor Page is associated with:

```
@model PrivacyModel
```

In this case the `PageModel` is called `PrivacyModel`, and it follows the standard convention for naming Razor Page models. You can find this class in the Privacy.cshtml.cs file in the Pages folder of your project, as shown in figure 2.15. Visual Studio nests these files underneath the Razor Page .cshtml files in Solution Explorer. We'll look at the page model in the next section.

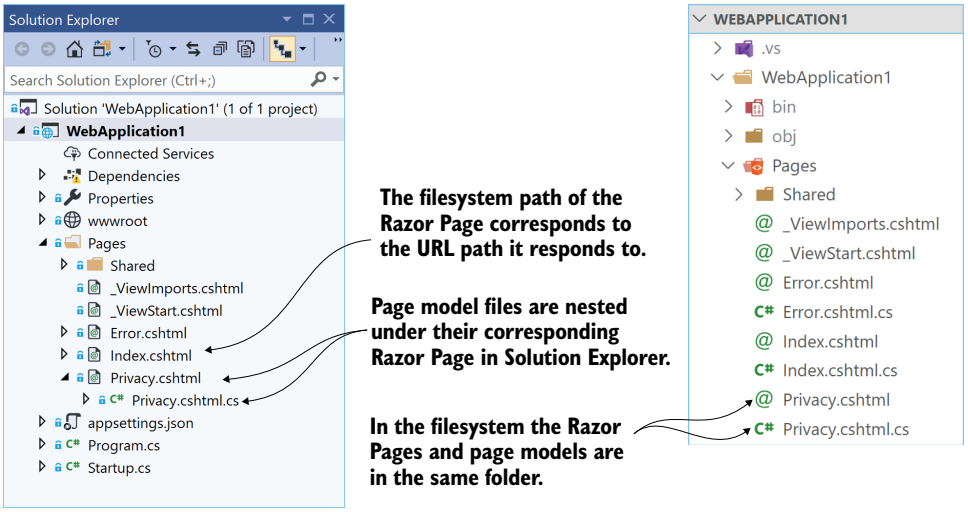

Figure 2.15 By convention, page models for Razor Pages are placed in a file with the same name as the Razor Page with a .cs suffix appended. Visual Studio nests these files under the Razor Page in Solution Explorer.

In addition to the `@page` and `@model` directives, you can see that static HTML is always valid in a Razor Page and will be rendered "as is" in the response.

```
<p>Use this page to detail your site's privacy policy.</p>
```

You can also write ordinary C# code in Razor templates by using this construct:

```
@{ /* C# code here */ }
```

Any code between the curly braces will be executed but won't be written to the response. In the listing, you're setting the title of the page by writing a key to the `ViewData` dictionary, but you aren't writing anything to the response at this point:

```
@{
    ViewData["Title"] = "Privacy Policy";
}
```

Another feature shown in this template is that you can dynamically write C# variables to the HTML stream using the `@` symbol. This ability to combine dynamic and static markup is what gives Razor Pages their power. In the example, you're fetching the `"Title"` value from the `ViewData` dictionary and writing the values to the response inside an `<h1>` tag:

```
<h1>@ViewData["Title"]</h1>
```

At this point, you might be a little confused by the template in listing 2.7 when it's compared to the output shown in figure 2.14. The title and the static HTML content appear in both the listing and figure, but some parts of the final web page don't appear in the template. How can that be?

Razor Pages have the concept of layouts, which are "base" templates that define the common elements of your application, such as headers and footers. The HTML of the layout combines with the Razor Page template to produce the final HTML that's sent to the browser. This prevents you from having to duplicate code for the header and footer in every page, and it means that, if you need to tweak something, you'll only need to do it in one place.

> **NOTE** I'll cover Razor templates, including layouts, in detail in chapter 7. You can find layouts in the Pages/Shared folder of your project.

As you've already seen, you can include C# code in your Razor Pages by using curly braces `@{ }`, but generally speaking you'll want to limit the code in your .cshtml file to presentational concerns only. Complex logic, code to access services such as a database, and data manipulation should be handled in the `PageModel` instead.

2.8.2 Handling request logic with PageModels and handlers

As you've already seen, the `@page` directive in a .cshtml file marks the page as a Razor Page, but most Razor Pages also have an associated *page model*. By convention, this is placed in a file commonly known as a "code behind" file that has a .cs extension, as you saw in figure 2.15. Page models should derive from the `PageModel` base class, and they typically contain one or more methods called *page handlers* that define how to handle requests to the Razor Page.

DEFINITION A *page handler* is a method that runs in response to a request. Razor Page models must be derived from the `PageModel` class. They can contain multiple page handlers, though typically they only contain one or two.

The following listing shows the page model for the Privacy.cshtml Razor Page, found in the file Privacy.cshtml.cs.

Listing 2.8 The `PrivacyModel` in Privacy.cshtml.cs—a Razor Page page model

```
public class PrivacyModel: PageModel
{
    private readonly ILogger<PrivacyModel> _logger;
    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
    }
}
```

Razor Pages must inherit from PageModel.

You can use dependency injection to provide services in the constructor.

The default page handler is OnGet. Returning void indicates HTML should be generated.

This page model is extremely simple, but it demonstrates a couple of important points:

- Page handlers are driven by convention.
- Page models can use dependency injection to interact with other services.

Page handlers are typically named by convention, based on the *HTTP verb* that they respond to. They return either `void`, indicating that the Razor Page's template should be rendered, or an `IActionResult` that contains other instructions for generating the response, such as redirecting the user to a different page.

DEFINITION Every HTTP request includes a *verb* that indicates the "type" of the request. When browsing a website, the default verb is `GET`, which *fetches* a resource from the server so you can view it. The second most common verb is `POST`, which is used to *send* data to the server, such as when completing a form.

The `PrivacyModel` contains a single handler, `OnGet`, which indicates it should run in response to `GET` requests for the page. As the method returns `void`, executing the handler will execute the associated Razor template for the page to generate the HTML.

NOTE Razor Pages are focused on building page-based apps, so you typically want to return HTML rather than JSON or XML. However, you can also use an `IActionResult` to return any sort of data, to redirect users to a new page, or to send an error. You'll learn more about `IActionResults` in chapter 4.

Dependency injection is used to inject an `ILogger<PrivacyModel>` instance into the constructor of the page model. The service is unused in this example, but it can be used to record useful information to a variety of destinations, such as the console, a file, or a remote logging service. You can access additional services in your page model by accepting them as parameters in the constructor—the ASP.NET Core framework will take care of configuring and injecting instances of any services you request.

> **NOTE** I describe the dependency inversion principle and the IoC container used in ASP.NET Core in detail in chapter 10. Logging is covered in chapter 17.

Clearly, the `PrivacyModel` page model does not do much in this case, and you may be wondering why it's worth having. If all they do is tell the Razor Page to generate HTML, then why do we need page models at all?

The key thing to remember here is that you now have a framework for performing arbitrarily complex functions in response to a request. You could easily update the handler method to load data from the database, send an email, add a product to a basket, or create an invoice—all in response to a simple HTTP request. This extensibility is where a lot of the power in Razor Pages (and the MVC pattern in general) lies.

The other important point is that you've separated the execution of these methods from the generation of the HTML itself. If the logic changes and you need to add behavior to a page handler, you don't need to touch the HTML generation code, so you're less likely to introduce bugs. Conversely, if you need to change the UI slightly, change the color of the title for example, then your handler method logic is safe.

And there you have it, a complete ASP.NET Core Razor Pages application! Before we move on, let's take one last look at how our application handles a request. Figure 2.16 shows a request to the `/Privacy` path being handled by the sample application. You've seen everything here already, so the process of handling a request should be familiar. It shows how the request passes through the middleware pipeline before being handled by the endpoint middleware. The Privacy.cshtml Razor Page executes the `OnGet` handler and generates the HTML response, which passes back through the middleware to the ASP.NET Core web server before being sent to the user's browser.

It's been a pretty intense trip, but you now have a good overview of how an entire application is configured and how it handles a request using Razor Pages. In the next chapter, you'll take a closer look at the middleware pipeline that exists in all ASP.NET Core applications. You'll learn how it's composed, how you can use it to add functionality to your application, and how you can use it to create simple HTTP services.

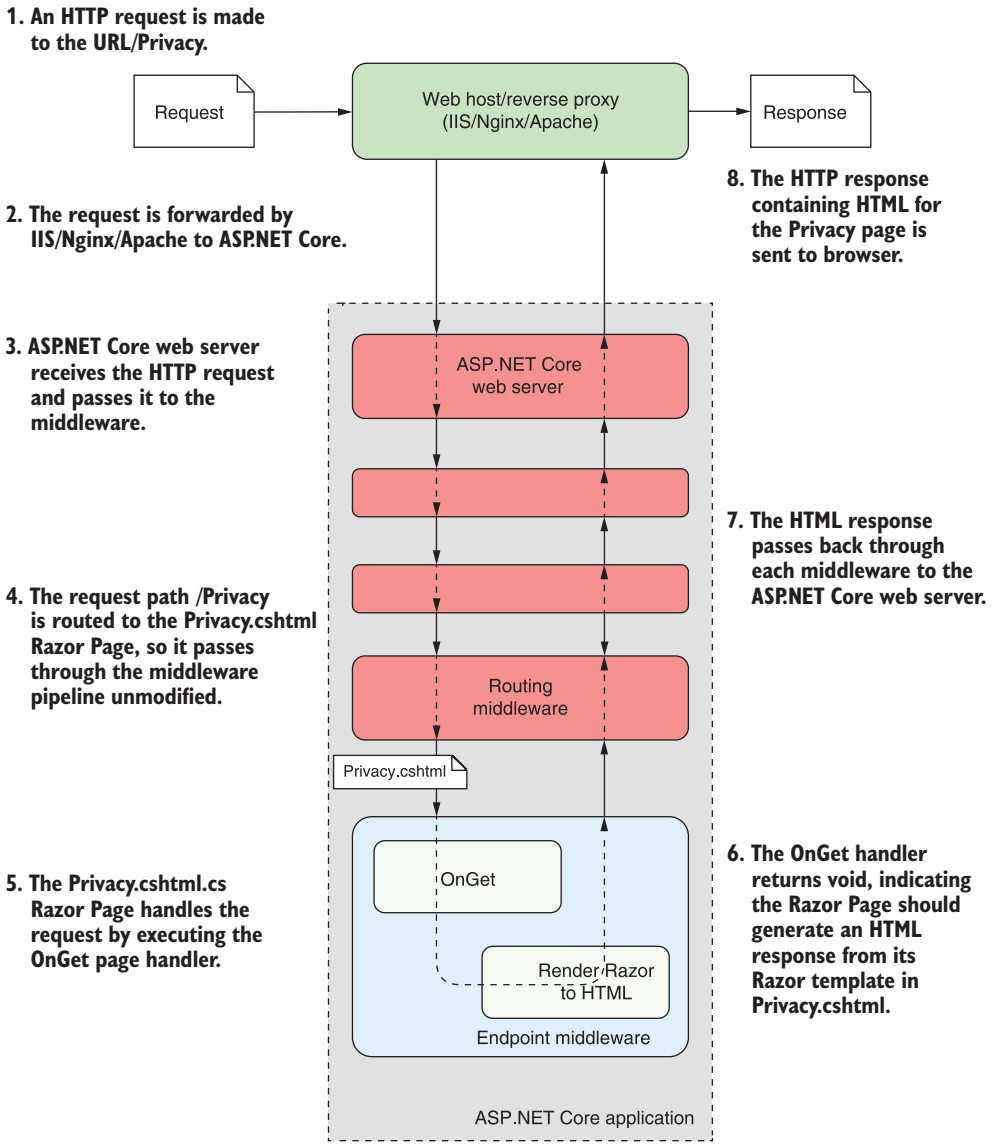

Figure 2.16 An overview of a request to the `/Privacy` URL for the sample ASP.NET Razor Pages application. The routing middleware routes the request to the `OnGet` handler of the Privacy.cshtml.cs Razor Page. The Razor Page generates an HTML response by executing the Razor template in Privacy.cshtml and passes the response back through the middleware pipeline to the browser.

Summary

- The .csproj file contains details of how to build your project, including which NuGet packages it depends on. It's used by Visual Studio and the .NET CLI to build your application.
- Restoring the NuGet packages for an ASP.NET Core application downloads all your project's dependencies so it can be built and run.
- Program.cs defines the `static void Main` entry point for your application. This function is run when your app starts, the same as for console applications.
- Program.cs is where you build an `IHost` instance, using an `IHostBuilder`. The helper method, `Host.CreateDefaultBuilder()` creates an `IHostBuilder` that loads configuration settings and sets up logging. Calling `Build()` creates the `IHost` instance.
- The `ConfigureWebHostDefaults` extension method configures the generic host using a `WebHostBuilder`. It configures the Kestrel HTTP server, adds IIS integration if necessary, and specifies the application's `Startup` class.
- You can start the web server and begin accepting HTTP requests by calling `Run` on the `IHost`.
- `Startup` is responsible for service configuration and defining the middleware pipeline.
- All services, both framework and custom application services, must be registered in the call to `ConfigureServices` in order to be accessed later in your application.
- Middleware is added to the application pipeline with `IApplicationBuilder`. Middleware defines how your application responds to requests.
- The order in which middleware is registered defines the final order of the middleware pipeline for the application. Typically, `EndpointMiddleware` is the last middleware in the pipeline. Earlier middleware, such as `StaticFileMiddleware`, will attempt to handle the request first. If the request is handled, `EndpointMiddleware` will never receive the request.
- Razor Pages are located in the Pages folder and are typically named according to the URL path they handle. For example, Privacy.cshtml handles the path `/Privacy`.
- Razor Pages must contain the `@page` directive as the first line of the file.
- Page models derive from the `PageModel` base class and contain page handlers. Page handlers are methods named using conventions that indicate the HTTP verb they handle. For example, `OnGet` handles the `GET` verb.
- Razor templates can contain standalone C#, standalone HTML, and dynamic HTML generated from C# values. By combining all three, you can build highly dynamic applications.
- Razor layouts define common elements of a web page, such as headers and footers. They let you extract this code into a single file, so you don't have to duplicate it across every Razor template.